# Домашняя работа по архитектуре ЭВМ

Рузавин Михаил М3137

29 декабря 2020 г.

Алгоритм транспозиции матрицы является одним из алгоритмов, на который влияет реализация кеша в компьютере. Поэтому для него существует реализация cache-oblivious алгоритма.

Ниже приведён алгоритм наивной реализации. Время работы - $O(nm)$.

```
fn transpose(&self) -> Matrix { //Matrix - structure containing size and elements.
    let mut result = Matrix::new(self.m, self.n); //Create new matrix size MxN
    for i in 0..self.n {
        for j in 0..self.m {
            result.matrix[j][i] = self.matrix[i][j];
        }
    }
    result
}
```

Для наивного алгоритма количество кеш промахов будет равно $\Theta(nm)$. Для ускорения программы необходимо уменьшить количество кеш промахов. Для этого можно использовать другой алгоритм, основная идея которого разделяй и властвуй, для такого алгоритма количество кеш промахов будет равно $O(mn/B)$ [1], где $B$ - размер кеша в элементах, а время работы останется неизменным $O(nm)$.

```
fn fast_transpose(&self) -> Matrix {
    let mut result = Matrix::new(self.m, self.n);
    self.fast_transpose_with_recursion(0, 0, self.n, self.m, &mut result);
    result
}
fn fast_transpose_with_recursion(&self, x: usize, y: usize, dx: usize, dy: usize,
                                                                out: &mut Matrix) {
    //x, y - submatrix start
    //dx, dy - submatrix size
    //out - transposed matrix (size: MxN)

    if dx == 1 && dy == 1 {
        out.matrix[y][x] = self.matrix[x][y];
    } else if dx >= dy {
        let mid = dx / 2;
        self.fast_transpose_with_recursion(x, y, mid, dy, out);
        self.fast_transpose_with_recursion(x + mid, y, dx - mid, dy, out);
    } else {
        let mid = dy / 2;
        self.fast_transpose_with_recursion(x, y, dx, mid, out);
        self.fast_transpose_with_recursion(x, y + mid, dx, dy - mid, out);
    }
}
```

[1] https://en.wikipedia.org/wiki/Cache-oblivious_algorithm

Времена алгоритмов были замерены[2] на квадратных матрицах. Ниже приведёны два сравнительных графика:

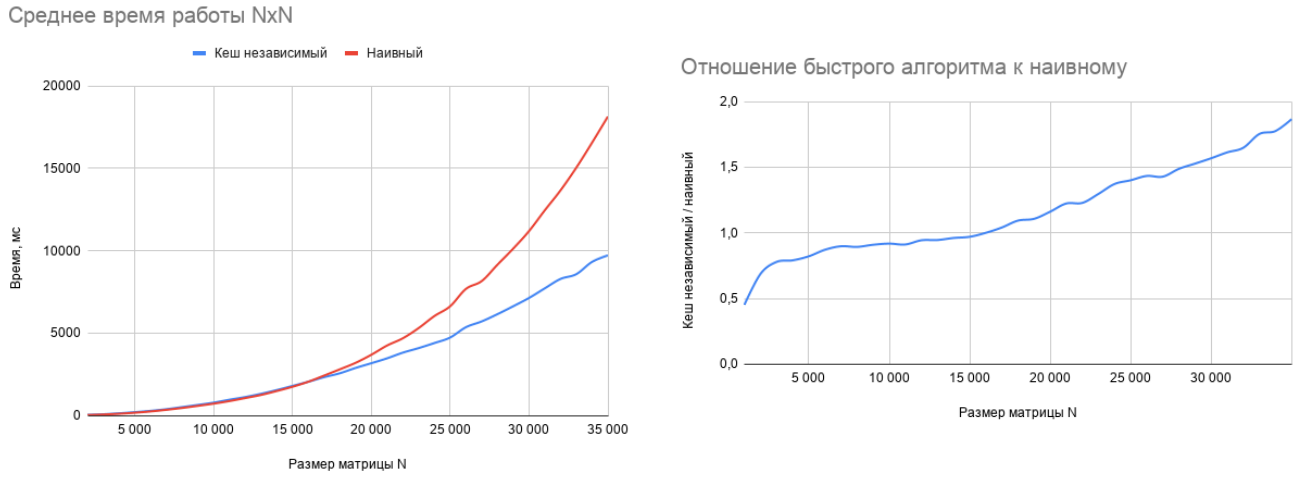


Рис. 1: Среднее время работы и отношение скорости cache-oblivious реализации к скорости наивной на матрице $N \times N$

На графиках можно заметить, что при размере матрицы примерно $17000 \times 17000$ cache-oblivious алгоритм начинает работать быстрее наивного, а на матрице размера $35000 \times 35000$ работает почти в два раза быстрее.

Также были произведены замеры на матрице с одним фиксированным размером $N \times 5000$, где заметно, что наивный алгоритм работает медленнее, чем cache-oblivious:

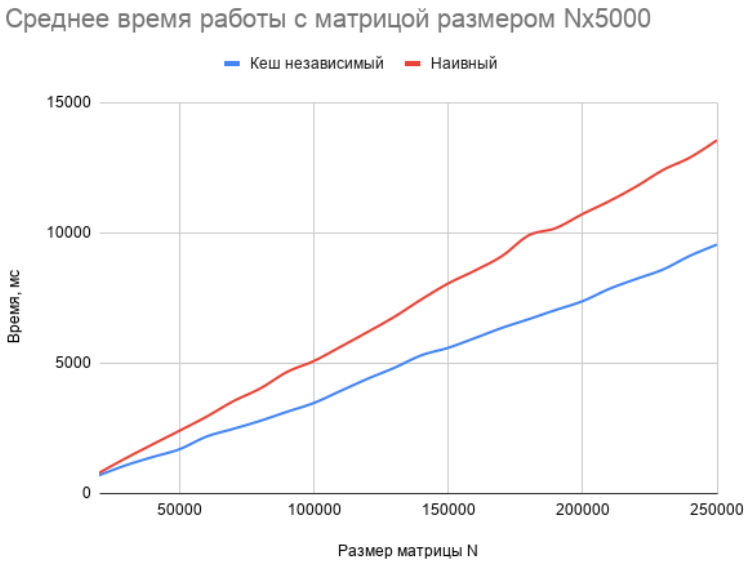


Рис. 2: График среднего времени исполнения наивного и cache-oblivious алгоритма на матрице размера $N \times 5000$

[2]Все замеры были произведены встроенными в Rust средствами. В замер времени входят: создании обёртки для транспонированной матрицы (пустая матрица размера $M \times N$) и её заполнение, используя некую исходную матрицу (создаётся вне таймера). Так же код был оптимизирован компилятором

Разница во времени исполнения алгоритмов связана, как уже писалось выше, с уменьшением числа кеш-промахов, что позволяет уменьшить количество обращений напрямую к памяти, время обращения к которой занимает больше времени чем к кешу.

Найти исходный код можно на GitHub: `https://github.com/mirout/CacheObliviousTransposeAlgorithm`. `main.rs` - содержит код, который использовался для замеров. `lib.rs` - содержит код для создания матрицы и её транспозиции. `transpose(&self)` - метод для запуска наивной транспозиции, возвращает новую транспонированную матрицу. `fast_transpose(&self)` - метод для запуска cache-oblivious транспозиции, также возвращает новую матрицу.

Для запуска кода нужно склонировать код с git, установить компилятор Rust и пакетный менеджер Cargo (можно воспользоваться rustup, который установит всё сам). Для запуска нужно открыть директорию, где лежит файл Cargo.toml, использовать команду `cargo run --release`.